Progetto SCPA

Implementazione e analisi di nuclei per prodotto SpMM

Gabriele Quatrana 0306403

AA 2022/2023

Indice

1 Introduzione

Questo documento descrive il processo di realizzazione di un nucleo di calcolo per il prodotto ***SpMM*** (*Sparse Matrix-Multivector*) in grado di calcolare:

$$Y = AX$$

dove A è una matrice sparsa memorizzata in due possibili formati:

- *CSR*
- *ELLPACK*

Mentre il multivettore X è una matrice densa di dimensione $n \cdot k$ con k piccolo.

I vari nuclei sono stati sviluppati in linguaggio *C* utilizzando le librerie di parallelizzazione *OpenMP*, nel caso CPU, e *CUDA Toolkit*, nel caso GPU. È stata realizzata una versione seriale del nucleo necessaria per confrontare i risultati con la versione parallela.

Oltre all'implementazione del programma, sono state analizzate le performance dei nuclei attraverso delle opportune metriche. Inoltre, sarà disponibile una piccola discussione finale sui risultati ottenuti durante lo svolgimento del progetto.

2 Moltiplicazione tra matrice sparsa e multivettore

Il problema della moltiplicazione *SpMM* è molto frequente in diversi ambiti informatici. Per realizzare il progetto, è stato sfruttato il fatto che la matrice sia sparsa, ciò significa che la maggior parte dei calcoli includerà dei valori nulli (ovvero pari a 0). È possibile considerare solo i valori non-zero e gli elementi del multivettore per realizzare i calcoli.

3 Gestione dei dati

I diversi nuclei fanno utilizzo di matrice memorizzate all'interno di file *MatrixMarket* ottenibili sul seguente sito `https://sparse.tamu.edu/`. Per scaricare le matrici di test è stato realizzato un piccolo script in *Bash*, chiamato `download_matrix.sh`. Per leggere tali file è stata utilizzata la libreria *Matrix Market I/O* disponibile all'indirizzo `https://math.nist.gov/MatrixMarket/`.

Le informazioni presenti nei file vengono salvate all'interno di una `struct` apposita chiamata `SparseMatrix`. Le matrici *MatrixMarket* possono essere di diverse tipologie, ma il codice implementato è in grado di gestirne due di queste:

- Le matrici *pattern* in cui non vengono indicati i valori non-zero che la compongono: in questo caso tutti i valori presenti in `SparseMatrix` vengono posti a 1.
- Le matrici *symmetric* in cui viene indicata solo la diagonale principale e la parte superiore della matrice: è necessario ricostruire la parte inferiore della matrice invertendo l'indice di riga con l'indice di colonna per gli elementi non facente parte della diagonale principale.

3.1 Formato CSR

Il formato *CSR* (*Compressed Storage by Rows*) memorizza i seguenti dati:

- M: numero di righe.
- N: numero di colonne.
- *IRP[M+1]*: vettore dei puntatori al primo elemento non-zero di ciascuna riga. *IRP[i]* contiene l'indice del primo elemento non-zero della riga i-esima.
- *JA[NZ]*: vettore degli indici di colonna di ciascun elemento. *JA[k]* contiene l'indice di colonna del k-esimo elemento non-zero.
- *AS[NZ]*: vettore dei coefficienti per ogni elemento non-zero della matrice. *AS[k]* contiene il valore del k-esimo elemento non-zero.

Il primo vettore da definire è *IRP*, che viene costruito nel seguente modo:

```
IRP[0] = 0;
for (int i = 0; i < matrix->M; i++) {
    IRP[i + 1] = IRP[i] + counters[i];
}
```

Viene usato un vettore ausiliario chiamato `counters` che mantiene, per ogni riga, il numero di valori non-zero della riga. In seguito è possibile popolare i vettori *JA* e *AS* attraverso *IRP*.

3.2 Formato ELLPACK

Il formato *ELLPACK* memorizza i seguenti dati:

- M: numero di righe.
- N: numero di colonne.
- *MAXNZ*: numero massimo di elementi non-zero tra tutte le righe.
- *JA[M][MAXNZ]*: array bidimensionale che contiene l'indice di colonna di ogni elemento non-zero per ogni riga della matrice. *JA[i][k]* contiene l'indice di colonna del k-esimo elemento non-zero dell'i-esima riga.
- *AS[M][MAXNZ]*: array bidimensionale che contiene il valore di ogni elemento non-zero per ogni riga della matrice. *AS[i][k]* contiene il valore del k-esimo elemento non-zero dell'i-esima riga.

In generale non tutte le righe della matrice hanno *MAXNZ* valori non-zero, bisogna quindi gestire questi casi andando a riempire in modo appropriato i vettori *JA* e *AS*:

- Per *AS* aggiungiamo valori **0** fino a raggiungere la lunghezza massima dell'array.
- Per *JA* replichiamo l'indice di colonna dell'ultimo valore non-zero della riga considerata.

4 Nuclei di Calcolo

Per l'implementazione della moltiplicazione *SpMM* sono stati realizzati due nuclei seriali e quattro nuclei paralleli in base al formato di memorizzazione delle matrici e al componente hardware utilizzato (CPU o GPU).

4.1 Nucleo Seriale

I nuclei di calcolo seriale sono stati implementati per valutare la correttezza del risultato ottenuto dai nuclei paralleli e per valutare la differenza di prestazioni (*speedup*) tra i due tipi di nucleo. Sono stati implementati due kernel, in base al formato di memorizzazione.

4.1.1 Formato CSR

Il nucleo di calcolo seriale per matrici memorizzate con il formato *CSR* è il seguente:

```
double result;
for (int row = 0; row < M; row++) {
    for (int k = 0; k < n; k++) {
        result = 0.0;
        for (int i = matrix->IRP[row]; i < matrix->IRP[row+1]; i++) {
            double matrix_val = matrix->AS[i];
            double vector_val = vector->val[matrix->JA[i] * n + k];
            result += matrix_val * vector_val;
        }
        val[row * n + k] = result;
    }
}
```

È stata utilizzata la variabile aggiuntiva `result` in modo da non aggiornare `val[row * n + k]` ad ogni iterazione ma solo alla fine del ciclo interno.

4.1.2 Formato ELLPACK

Il nucleo di calcolo seriale per matrici memorizzate con il formato *ELLPACK* è il seguente:

```
double result;
for (int i = 0; i < M; i++) {
    for (int k = 0; k < n; k++) {
        int row = i * MAXNZ;
        result = 0.0;
        for (int j = 0; j < MAXNZ; j++) {
            double matrix_val = matrix->AS[row+j];
            double vector_val = vector->val[matrix->JA[row+j] * n + k];
            result += matrix_val * vector_val;
        }
        val[i * n + k] = result;
    }
}
```

Anche in questo caso è stata utilizzata la variabile `result` per lo stesso scopo del caso *CSR*.

4.2 Nucleo Parallelo

Per il calcolo parallelo sono stati realizzati quattro nuclei divisi in base al componente hardware che viene utilizzato e al formato di memorizzazione delle matrici.

4.2.1 Nucleo CPU

Per il calcolo parallelo basato su CPU è stata utilizzata la libreria *OpenMP* per distribuire le righe delle matrici tra i vari thread del processore della macchina.

4.2.1.1 Formato CSR

Il nucleo di calcolo parallelo CPU per matrici memorizzate con il formato *CSR* è il seguente:

```
int M = matrix->M;
int nzPerRow = nz / M;
int chunk_size = NZ_PER_CHUNK / nzPerRow;
...
int i, j, k, tmp;
#pragma omp parallel default(none) shared(M, n, chunk_size, matrix, vector, val)
{
    #pragma omp for private(i, j, k, tmp) schedule(dynamic, chunk_size)
    for (i = 0; i < M; i++) {
        for (k = 0; k < n; k++) {
            double result = 0.0;
            #pragma omp simd reduction(+ : result)
            for (j = matrix->IRP[i]; j < matrix->IRP[i+1]; j++) {
                result += matrix->AS[j] * vector->val[matrix->JA[j] * n + k];
            }
            val[i * n + k] = result;
        }
    }
}
```

Come nel caso seriale è stata utilizzata la variabile `result` per evitare di aggiornare ad ogni iterazione del ciclo interno il valore `val[i * n + k]`.

Il ciclo esterno sulle righe è stato suddiviso sui vari thread utilizzando uno schedule dinamico con un valore `chunk_size` calcolato appositamente per ogni matrice. In questo modo ogni chunk include in media un numero di valori non-zero pari a `NZ_PER_CHUNK`. La scelta tra uno schedule dinamico e uno statico dovrebbe dipendere dal tipo di matrice:

- Lo schedule dinamico risulta essere migliore nel caso in cui le matrici hanno un numero molto variabile di valori non-zero per riga in quanto permette di bilanciare maggiormente il carico tra i vari thread, introducendo però un alto overhead.
- Lo schedule statico risulta essere migliore nel caso in cui le matrici hanno un numero poco variabile di valori non-zero per riga in quanto il bilanciamento del carico viene realizzato automaticamente senza overhead aggiuntivo.

Per il ciclo interno è stata applicata la vettorizzazione per velocizzare l'esecuzione del kernel. La vettorizzazione del ciclo è stata implementata tramite la direttiva `#simd`. Il processore è in grado di supportare la vettorizzazione se è provvisto del set di istruzioni *AVX* (*Advanced Vector Extension*). Per verificare la loro presenza, è possibile utilizzare il comando `grep avx /proc/cpuinfo`: se viene restituito un risultato a schermo allora il processore supporta la vettorizzazione altrimenti non ha il set *AVX*. La vettorizzazione del ciclo interno di questa implementazione ha comunque due criticità [1]:

- **Loop Remainder**: un'implementazione vettorizzata del kernel deve gestire delle iterazioni rimanenti del ciclo nel caso in cui il numero di valori non-zero in una riga non sono un multiplo della lunghezza del vettore *SIMD*. Questa criticità diventa più importante nel caso in cui ogni riga ha solo pochi valori non-zero. In generale, per avere un'esecuzione efficiente è necessario che il numero di non-zeri per riga sia divisibile per la lunghezza del vettore *SIMD*.
- **Data Locality**: il ciclo interno itera sulle righe che, grazie al formato di memorizzazione, sono salvate consecutivamente in memoria per cui l'accesso agli elementi della matrice è sequenziale.

Tuttavia, l'accesso agli elementi del vettore dipende dagli indici delle colonne degli elementi non-zero della matrice: se i non-zero sono distribuiti in modo sparso, anche l'accesso agli elementi del vettore seguirà un pattern casuale, di conseguenza non si può sfruttare la località dei dati.

4.2.1.2 Formato ELLPACK

Il nucleo di calcolo parallelo CPU per matrici memorizzate con il formato *ELLPACK* è il seguente:

```
int M = matrix->M;
...
int i, k, j, tmp, row;
#pragma omp parallel default(none) shared(M, n, MAXNZ, matrix, vector, val)
{
    #pragma omp for private(i, k, j, tmp, row) schedule(static)
    for (i = 0; i < M; i++) {
        for (k = 0; k < n; k++) {
            double result = 0.0;
            row = i * MAXNZ;
            #pragma omp simd reduction(+ : result)
            for (j = 0; j < MAXNZ; j++) {
                tmp = matrix->JA[row+j];
                result += matrix->AS[row + j] * vector->val[tmp * n + k];
            }
            val[i * n + k] = result;
        }
    }
}
```

In questo caso il ciclo esterno sulle righe è stato suddiviso tra i thread del processore utilizzando uno schedule statico; infatti, il bilanciamento del carico è già fornito dal formato *ELLPACK*: tutte le righe, dopo il padding, hanno la stessa lunghezza, quindi non è necessario introdurre un overhead per bilanciare manualmente il carico.

Come nel caso precedente, per il ciclo interno è stata abilitata la vettorizzazione tramite la direttiva `#simd`: rimangono comunque le criticità di *Loop Remainder* e *Data Locality*.

4.2.2 Nucleo GPU

Per il calcolo parallelo basato su GPU è stata utilizzata la libreria *CUDA Toolkit* per eseguire porzioni di codice direttamente sui thread della scheda video presente sulla macchina.

4.2.2.1 Formato CSR

Per il calcolo parallelo basato su GPU con matrici *CSR* è stato utilizzato l'algoritmo *CSR-Adaptive* che consente di stabilire durante l'esecuzione se usare *CSR-Vector* oppure *CSR-Stream*. L'idea di base del *CSR-Adaptive* è la seguente:

- Se una riga della matrice è "lunga" su questa viene eseguito *CSR-Vector*: la riga viene assegnata ad un *warp*.
- Se si hanno più righe della matrice "corte" viene eseguito *CSR-Stream*: le righe vengono raggruppate in un insieme di righe che viene assegnato a un blocco di thread.

Definiamo come righe "corte" quelle che hanno un numero di valori non-zero abbastanza piccolo tale che l'algoritmo *CSR-Vector* non ha delle buone prestazioni. Al contrario, le righe "lunghe" sono quelle che hanno un numero di valori non-zero abbastanza grande tale che l'algoritmo *CSR-Vector* ha delle buone prestazioni.

Risulta necessario, quindi, calcolare i blocchi di righe ovvero gli insiemi di righe contigue che ogni blocco può gestire. Per determinare tali blocchi è stato implementato l'**algoritmo 2** dell'articolo [2]:

```
int *find_row_blocks(int totalRows, const int *IRP, int *block_count) {

    int *row_blocks = (int *) calloc(MAX_GRID_SIZE, sizeof(int));
    row_blocks[0] = 0;

    int last_i = 0; // Last row added to a row block
    int ctr = 1;    // Current row block
    int sum = 0;    // Index of the NZ element trying to add to the row block

    for (int i = 1; i < totalRows; i++) {

        sum += IRP[i] - IRP[i-1];

        // The matrix row fit perfectly in the current row block
        if (sum == MAX_NZ_PER_ROW_BLOCK) {
            last_i = i;
            row_blocks[ctr++] = i;
            sum = 0;
        }

        // The matrix row does not fit into the current row block
        else if (sum > MAX_NZ_PER_ROW_BLOCK) {

            // Reconsider the row
            if (i - last_i > 1) {
                row_blocks[ctr++] = i - 1;
                i--;
            }

            // The row block contains one matrix row
            else if (i - last_i == 1) {
                row_blocks[ctr++] = i;
            }

            last_i = i;
            sum = 0;
        }
    }

    *block_count = ctr;
    row_blocks[ctr++] = totalRows;
    return row_blocks;
}
```

L'algoritmo scorre le varie righe della matrice e conta il numero di valori non-zero. Per ogni riga si hanno due possibilità:

1. Il numero di valori non-zero contati fino alla riga considerata è esattamente pari al massimo numero di non-zero per blocco di righe: il blocco di righe viene chiuso e si considera la riga successiva.
2. Il numero di valori non-zero contati fino alla riga considerata è maggiore rispetto al massimo numero di non-zero per blocco di righe:
   (a) Se i non-zero accumulati fanno parte di più righe il blocco di righe viene chiuso alla riga precedente e viene riconsiderata la riga corrente.
   (b) Se i non-zero accumulati fanno parte della stessa riga viene creato un blocco di righe che comprende solo la riga corrente.

L'algoritmo produce in output un'array che contiene gli indici delle righe in cui cominciano i diversi blocchi di righe, che è necessario per eseguire l'algoritmo *CSR-Adaptive*, il cui nucleo di calcolo è il seguente:

```
__global__ void csr_adaptive(int n,
                             const int *JA,
                             const int *IRP,
                             const double *AS,
                             const double *vec,
                             double *res_vec,
                             const int *row_blocks) {

    const int block_start = row_blocks[blockIdx.x];
    const int block_end = row_blocks[blockIdx.x + 1];
    const int num_rows = block_end - block_start;

    // CSR-Stream selected (the row block contains 2 or more rows)
    if (num_rows > 1) {
        csr_stream(num_rows, n, JA, IRP, AS, vec, res_vec, block_start, block_end);
    }

    // CSR-Vector selected (the row block contains only one row)
    else {
        csr_vector(n, JA, IRP, AS, vec, res_vec, block_row_start);
    }
}
```

Dall'implementazione è possibile vedere che se il blocco di righe contiene due o più righe viene eseguito *CSR-Stream*, altrimenti si esegue *CSR-Vector*. Questo significa che *CSR-Vector* viene eseguito sempre su una sola riga ("lunga") per volta della matrice. Il nucleo di calcolo per *CSR-Vector* è il seguente:

```
__device__ void csr_vector(int n,
                           const int *JA,
                           const int *IRP,
                           const double *AS,
                           const double *vec,
                           double *res_vec,
                           const int row) {

    unsigned int thread_id = threadIdx.x;
    const unsigned int warp_id = thread_id / 32;
    const unsigned int lane = thread_id % 32;

    int col;
    double val;
    double sum[64] = {0};

    if (warp_id < n) {
        for (unsigned int j = warp_id; j < n; j += 32) {
            for (unsigned int i = IRP[row] + lane; i < IRP[row + 1]; i += 32) {
                val = AS[i];
                col = JA[i];
                sum[j] += val * vec[col * n + j];
            }

            sum[j] = warp_reduce(sum[j]);

            if (lane == 0) {
                res_vec[row * n + j] = sum[j];
            }
        }
    }
}
```

L'algoritmo assegna ad ogni riga della matrice un *warp* e calcola il valore *lane* che rappresenta l'indice di un thread all'interno del *warp*. Ciascun thread calcola un risultato parziale relativo ai valori non-zero della riga il cui indice `i` soddisfa la condizione `i%32==lane`. Questi risultati vengono salvati in un array privato di dimensione pari al numero di colonne del multivettore.

Una volta calcolati i risultati parziali, viene eseguita una riduzione attraverso la seguente funzione:

```
__device__ double warp_reduce(double value) {
    for (int offset = warpSize/2; offset > 0; offset /= 2) {
        value += __shfl_down_sync(FULL_MASK, value, offset);
    }
    return value;
}
```

Questa riduzione sfrutta la primitiva a livello di *warp* `__shfl_down_sync()` che permette di effettuare una *tree-reduction* che risulta più efficiente di una riduzione effettuata in shared memory, come illustrato da *NVIDIA* [3].

CSR-Vector risulta, quindi, essere adatto a scenari in cui le righe della matrice sono "lunghe", ovvero hanno tanti elementi non-zero. Infatti, nel caso in cui si hanno righe "corte", le performance diminuiscono a causa della mancanza di lavoro parallelo (solo un sottoinsieme di thread effettua lavoro utile) e, di conseguenza, non viene ammortizzato il costo della riduzione.
Il nucleo di calcolo per *CSR-Stream* si basa sull'**algoritmo 3** dell'articolo [2]:

```
__device__ void csr_stream(int num_rows,
                           int n,
                           const int *JA,
                           const int *IRP,
                           const double *AS,
                           const double *vec,
                           double *res_vec,
                           const int block_row_start,
                           const int block_row_end) {

    __shared__ int shared_JA[MAX_NZ_PER_ROW_BLOCK];
    __shared__ double shared_AS[MAX_NZ_PER_ROW_BLOCK];

    unsigned int thread_id = threadIdx.x;
    unsigned int block_size = blockDim.x;
    int num_non_zeroes = IRP[block_row_end] - IRP[block_row_start];

    // Stream JA and AS into shared memory
    unsigned int local_col;
    for (unsigned int i = thread_id; i < num_non_zeroes; i += block_size) {
        local_col = IRP[block_row_start] + i;
        shared_JA[i] = JA[local_col];
        shared_AS[i] = AS[local_col];
    }
    int first_element_col = IRP[block_row_start];

    __syncthreads();

    for (unsigned int t = thread_id; t < num_rows * n; t += blockDim.x) {
        unsigned int matrix_row = block_row_start + t / n;
        unsigned int vector_col = t % n;
        double sum = 0.0;
        for (int i = IRP[matrix_row] - first_element_col;
             i < IRP[matrix_row + 1] - first_element_col;
             i++) {
            sum += shared_AS[i] * vec[shared_JA[i] * n + vector_col];
        }
        res_vec[matrix_row * n + vector_col] = sum;
    }

    __syncthreads();
}
```

Analizziamo l'algoritmo:

- Esegue lo stream in memoria condivisa (`shared memory`) degli array `JA` e `AS`. Si noti che si effettuano accessi coalescenti agli elementi della matrice in memoria globale.
- Ogni thread viene assegnato a una riga e a una colonna del multivettore, ed esegue il prodotto tra queste utilizzando i valori in memoria condivisa.
- I risultati vengono salvati nel multivettore di output.

Nel *CSR-Stream* i risultati parziali calcolati da ciascun thread non possono essere ridotti tramite una primitiva a livello di *warp*, come nel caso *CSR-Vector*, in quanto i risultati parziali appartengono a righe diverse della matrice e, inoltre, i risultati parziali di una stessa riga potrebbero essere calcolati da thread che appartengono a *warp* differenti dello stesso blocco.

Per questi motivi gli array `JA` e `AS` vengono prima memorizzati in memoria condivisa e poi viene eseguita la riduzione per ciascuna riga che compone il blocco di righe. Inoltre, il numero di riduzioni da effettuare e il numero di elementi da ridurre non è fissato, ma varia per i diversi blocchi di righe.

Per l'esecuzione del *CSR-Adaptive* sono stati usati blocchi **1D** posti in una griglia **1D**. In particolare sono stati usati blocchi di dimensione pari a **1024**. La dimensione della griglia è stata scelta pari al numero di blocchi di righe individuati e dipende, quindi, dal massimo numero di valori non-zero per blocco di righe:

```
const dim3 block_size = dim3(BLOCK_SIZE_CSR);
const dim3 grid_size = dim3(block_count);
```

4.2.2.2 Formato ELLPACK

Il nucleo di calcolo parallelo GPU per matrici memorizzate con il formato *ELLPACK* è il seguente:

```
__global__ void ell_kernel(int m,
                           int n,
                           int MAXNZ,
                           const int *JA,
                           const double *AS,
                           const double *vec,
                           double *res_vec) {

    unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < m * n) {
        unsigned int matrix_col = idx / n;
        unsigned int vector_col = idx % n;
        double sum = 0.0;
        for (int j = 0; j < MAXNZ; j++) {
            unsigned int matrix_index = matrix_col + m * j;
            double matrix_val = AS[matrix_index];
            double vector_val = vec[JA[matrix_index] * n + vector_col];
            sum += matrix_val * vector_val;
        }
        res_vec[idx] = sum;
    }
}
```

Ogni elemento del multivettore viene assegnato a ciascun thread e, a partire dal suo indice, viene calcolata la colonna del multivettore relativa e la colonna della matrice con la quale eseguire il prodotto. In questo caso, infatti, viene eseguita una trasposizione della matrice, ovvero degli array *JA* e *AS* del formato *ELLPACK*, e in questo modo gli elementi di ciascuna colonna sono adiacenti in memoria.

Per l'esecuzione del kernel è stata utilizzata una griglia **1D** di blocchi **1D**. I blocchi hanno una dimensione di **1024** thread e la dimensione della griglia è stata calcolata in modo da coprire tutte le righe della matrice nel seguente modo:

```
const dim3 block_size = dim3(BLOCK_SIZE_ELLPACK);
const dim3 grid_size = dim3(M+1);
```

5 Analisi delle prestazioni

Prima di misurare le prestazioni dei nuclei illustrati in precedenza, è stata verificata la correttezza dei risultati prodotti dai nuclei. Questi vengono confrontati con i risultati del prodotto seriale, valutando la differenza tra i risultati ottenuti: se il valore assoluto della differenza tra i risultati è minore di 10^{-7} allora il risultato prodotto viene considerato corretto, altrimenti viene restituito un errore.

Le prestazioni del codice prodotto sono state misurate attraverso degli appositi timer che permettono di calcolare il tempo necessario per eseguire il *SpMM*. Le matrici considerate per misurare le prestazioni del codice sono le seguenti:

cage4	Cube_Coup_dt0	FEM_3D_thermal1
mhda416	ML_Laplace	thermal1
mcfe	bcsstk17	thermal2
olm1000	mac_econ_fwd500	thermomech_TK
adder_dcop_32	mhd4800a	nlpkkt80
west2021	cop20k_A	webbase-1M
cavity10	raefsky2	dc1
rdist2	af23560	amazon0302
cant	lung2	af_1_k101
olafu	PR02R	roadNet-PA

Per misurare le performance dei nuclei prodotti, il codice è stato eseguito sul server di dipartimento dell'università di *Roma Tor Vergata* che dispone di due CPU *Intel Xeon Silver 4210* da *2.20GHz*, ognuna delle quali possiede 10 core, e di una GPU *Nvidia Quadro RTX 5000*.

Per ogni matrice il calcolo è stato ripetuto più volte considerando varie dimensioni dei multivettori in base al numero di colonne (3,4,8,12,16,32,64). Per ogni coppia matrice-multivettore il calcolo è stato ripetuto **10** volte ed è stata considerata la media tra tutte le esecuzioni. Le metriche utilizzate per valutare le prestazioni sono le seguenti:

- Il **tempo di calcolo** (in millisecondi) del prodotto *SpMM*.
- I **flops** (nello specifico *GigaFlops*).
- Lo **speedup** rispetto all'implementazione seriale.

Queste metriche vengono salvate, alla fine dell'esecuzione del programma, in un file `csv` in base al tipo di nucleo di calcolo utilizzato.

5.1 Analisi OpenMP

Per analizzare le performance dei nuclei di calcolo sulla CPU è stato fatto variare il numero di thread da un minimo di **1** a un massimo di **20**, pari al numero di core disponibili sulla macchina utilizzata per misurare le prestazioni.

5.1.1 Formato CSR

Le figure 1 e 2 mostrano i *GigaFlops* ottenuti utilizzando il formato *CSR* e un multivettore con, rispettivamente, 3 e 64 colonne al variare del numero di thread. Possiamo notare che per matrici grandi si ottengono prestazioni nettamente migliori all'aumentare del numero di thread utilizzati, mentre per le matrici più piccole, invece, non si ottengono miglioramenti in quanto sono necessari pochi calcoli per eseguire il prodotto: i thread generati sono sottoposti a un carico minimo che non giustifica il loro utilizzo.
Il grafico 3 mostra l'andamento medio dei *GigaFlops* con 20 thread al variare del numero di colonne del multivettore. Si può notare come per `k` più grandi si ottengono prestazioni, in generale, più elevate.

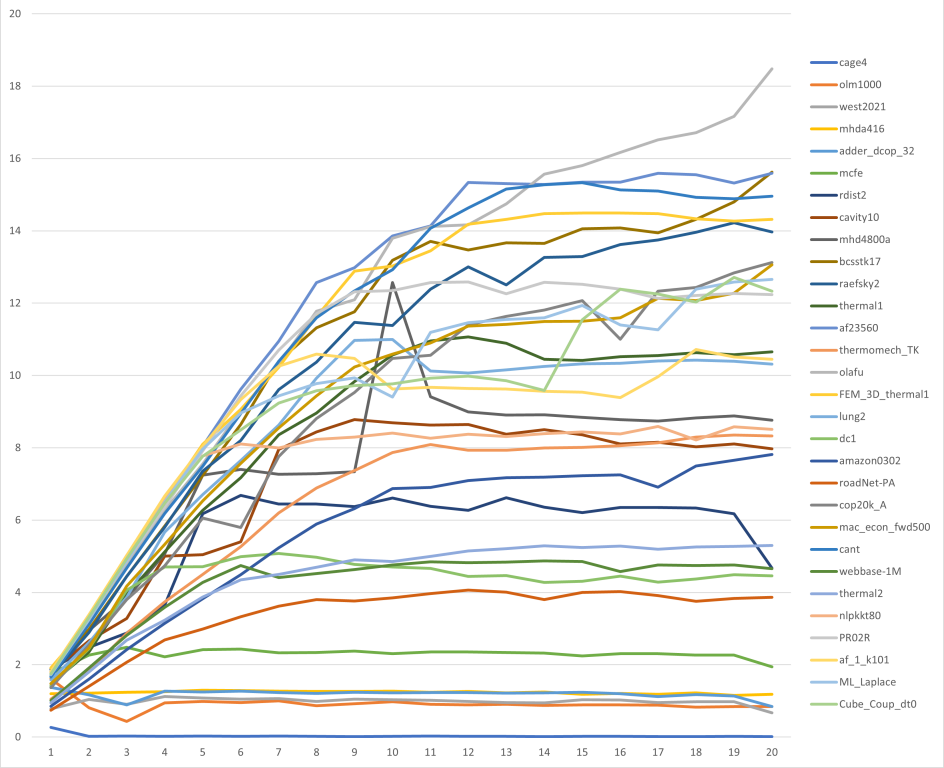

Chart 1: OpenMP CSR GFLOPS with k = 3

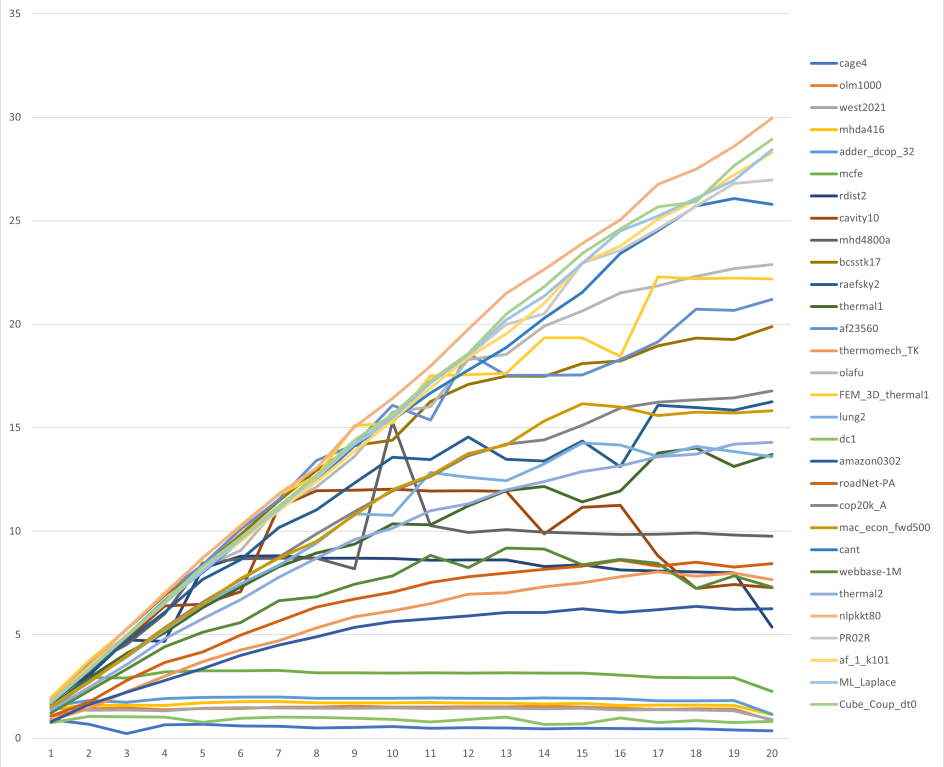

Chart 2: OpenMP CSR GFLOPS with k = 64

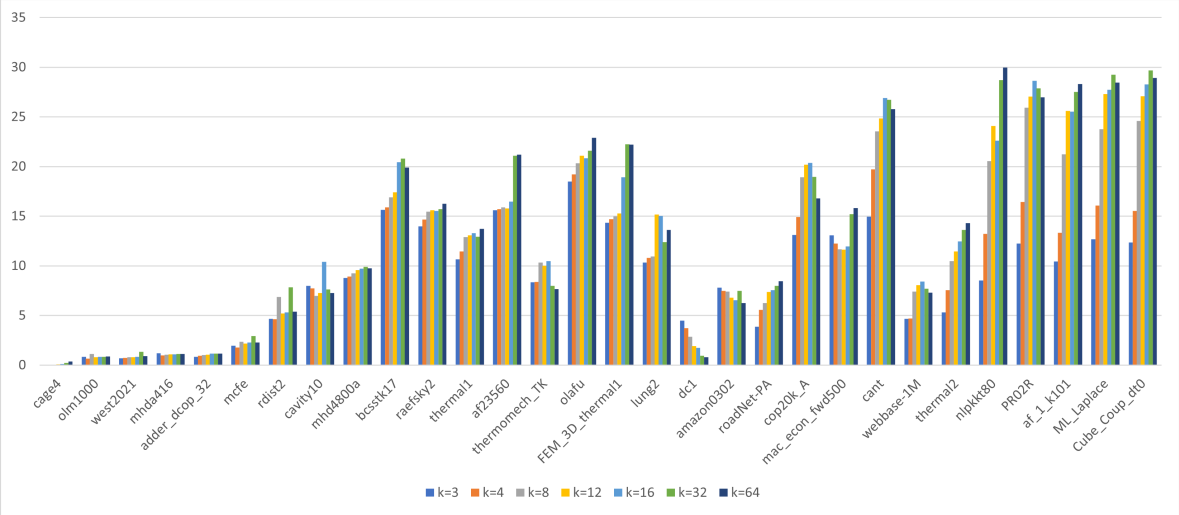

Chart 3: OpenMP CSR GFLOPS as k varies

Ci sono alcune eccezioni, come la matrice `dc1`, la cui particolarità è che ha alcune righe con molti più valori non-zero rispetto alle altre, di conseguenza ci sono alcuni thread che eseguono molto più lavoro rispetto agli altri e che rappresentano, quindi, un collo di bottiglia.

Il grafico 4, invece, mostra lo speedup medio con 20 thread al variare del numero di colonne del multivettore. Possiamo vedere che in generale le prestazioni sono migliori rispetto al prodotto seriale, soprattutto nel caso di valori `k` grandi. Sono presenti, però, alcune eccezioni:

- Matrici molto piccole (come `cage4`) hanno prestazioni di poco migliori o addirittura peggiori: questo è causato dall'overhead necessario per la creazione dei thread.
- Matrici che hanno alcune righe con molti più valori non-zero rispetto alle altre come abbiamo visto in precedenza: le prestazioni risultano comunque migliori rispetto al caso seriale.

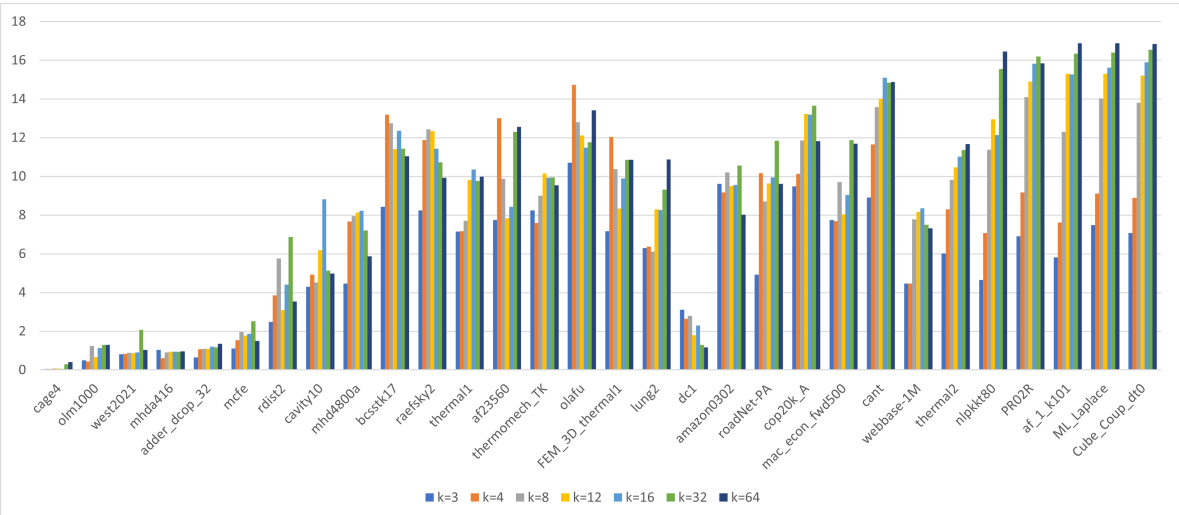

Chart 4: OpenMP CSR speedup as k varies

Nella tabella 1 sono contenuti i risultati numerici ottenuti in termini di tempo speso, *GigaFlops* e speedup.

Matrix Name	**Time (ms)**	**Flops (GF)**	**SpeedUp (%)**
cage4	0.0173	0.362543	40.4624
olm1000	0.5969	0.856907	129.67
west2021	1.0493	0.896964	103.116
mhda416	0.9622	1.13899	95.8221
adder_dcop_32	1.2441	1.15705	135.037
mcfe	1.3801	2.26135	149.409
rdist2	1.3561	5.3739	353.956
cavity10	1.3443	7.27142	498.996
mhd4800a	1.3402	9.7659	587.524
bcsstk17	2.7588	19.8881	1103.52
raefsky2	2.3174	16.2541	992.405
thermal1	5.3657	13.7038	998.118
af23560	2.9236	21.2015	1255.99
thermomech_TK	11.8931	7.65817	954.781
olafu	5.6771	22.8884	1340.9
FEM_3D_thermal1	2.4845	22.1915	1085.45
lung2	4.6354	13.6015	1086.81
dc1	120.519	0.813968	116.696
amazon0302	25.279	6.25279	802.765
roadNet-PA	46.7843	8.43714	962.361
cop20k_A	20.0272	16.7729	1181.95
mac_econ_fwd500	10.2994	15.8256	1168.8
cant	19.8893	25.79	1488.34
webbase-1M	54.4774	7.29676	732.19
thermal2	76.8278	14.2953	1167.16
nlpkkt80	122.679	29.9496	1645.56
PR02R	38.8384	26.9758	1584.76
af_1_k101	79.4159	28.2876	1688.94
ML_Laplace	124.627	28.4394	1688.13
Cube_Coup_dt0	562.884	28.9267	1683.91

Table 1: OpenMP CSR metrics with 20 threads and k = 64

5.1.2 Formato ELLPACK

Le figure 5 e 6 mostrano i *GigaFlops* ottenuti utilizzando il formato *CSR* e un multivettore con, rispettivamente, 3 e 64 colonne al variare del numero di thread. Come nel caso *CSR*, si ottengono prestazioni molto maggiori per matrici grandi, mentre, per le matrici piccole, non si ottengono miglioramenti. La differenza principale è data dall'andamento più irregolare rispetto al caso precedente: le prestazioni tendono a crescere linearmente fino a 10/11 thread per il caso `k=3` e fino a 13/14 thread per il caso `k=64` per poi avere diversi salti fino ad arrivare a 20 thread. Questo è dovuto alla natura del formato *ELLPACK* che introduce un overhead molto più importante rispetto al formato *CSR*.

Notiamo, inoltre, l'assenza delle matrici `webbase-1M` e `dc1` a causa del fatto che hanno un valore `MAXNZ` troppo elevato. Questo rende impossibile garantire il corretto funzionamento del programma a causa del numero di zeri aggiunti necessari per la memorizzazione degli array *JA* e *AS* del formato *ELLPACK* per queste matrici. Ignorare questo controllo può causare buffer overflow per le variabili nel momento in cui vengono calcolati gli indici (ad esempio `row = i * MAXNZ`), di conseguenza il comportamento del codice diventa imprevedibile e si possono ottenere errori di segmentazione.

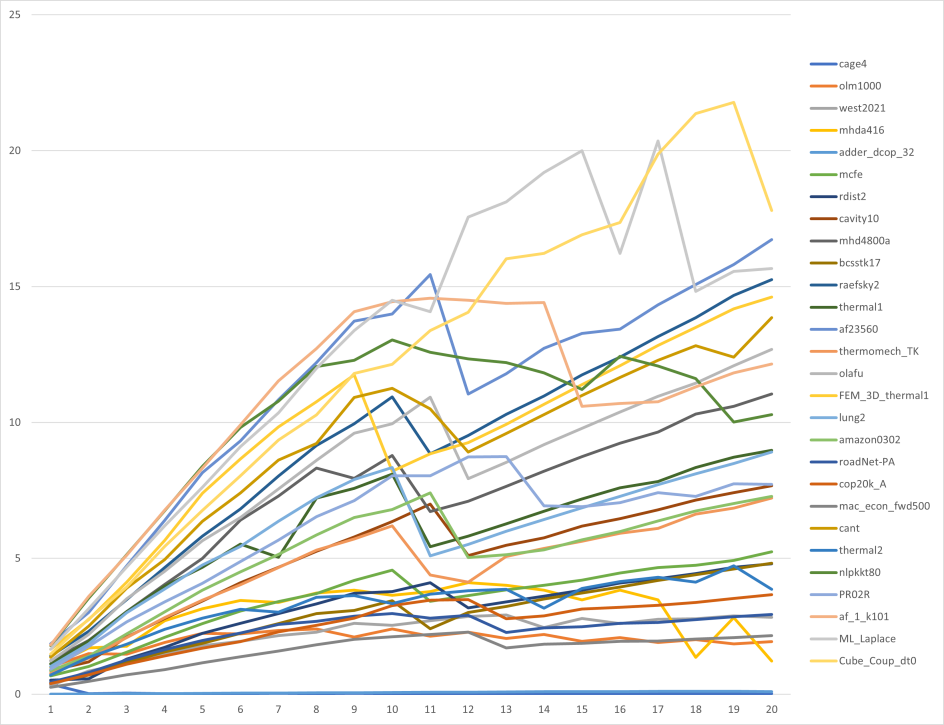

Chart 5: OpenMP ELLPACK GFLOPS with k = 3

Il grafico 7 mostra l'andamento medio dei *GigaFlops* con 20 thread al variare del numero di colonne del multivettore. Come nel caso *CSR*, per `k` più grandi si ottengono prestazioni più elevate. In generale, con il formato *CSR* si ottengono delle prestazioni migliori in termini di *GigaFlops*.

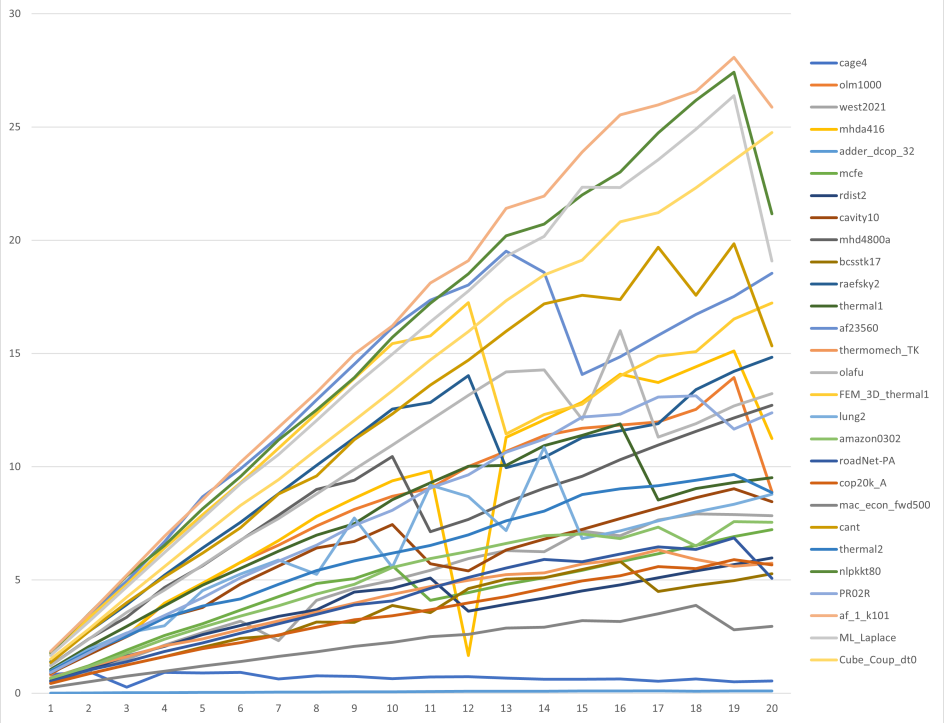

Chart 6: OpenMP ELLPACK GFLOPS with k = 64

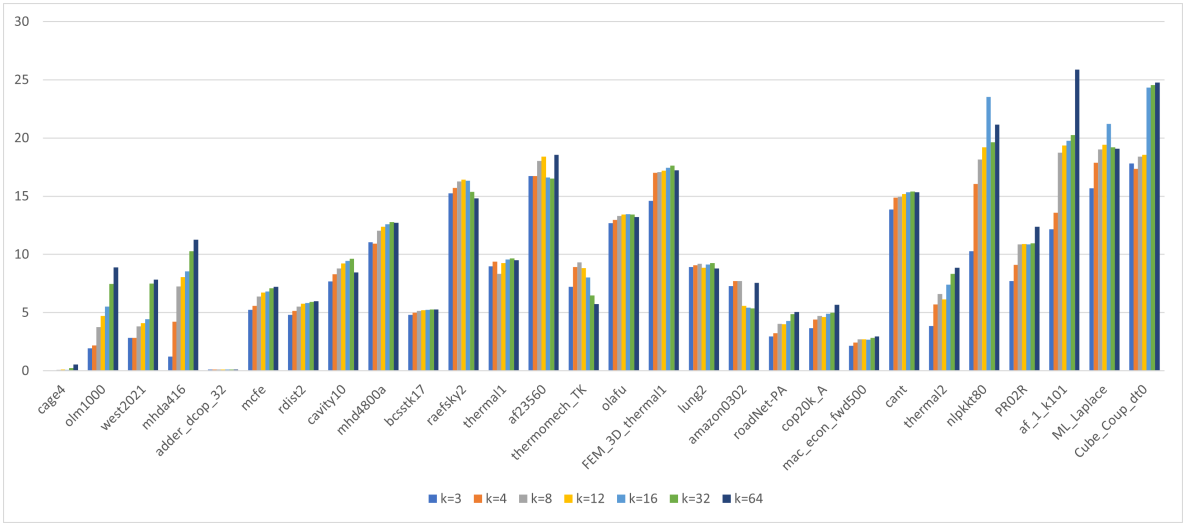

Chart 7: OpenMP ELLPACK GFLOPS as k varies

La figura 8 mostra, invece, lo speedup medio con 20 thread al variare del numero di colonne del multivettore. Possiamo trarre le stesse conclusioni ottenute nel caso *CSR* e notiamo che, in generale, quest'ultimo formato fornisce uno speedup più elevato rispetto a *ELLPACK*.

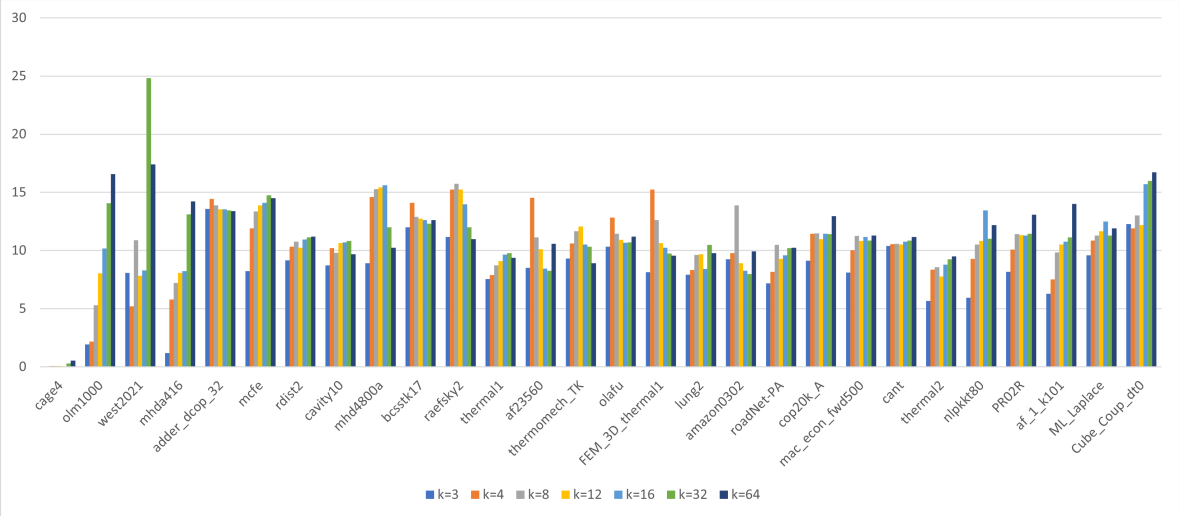

Chart 8: OpenMP ELLPACK speedup as k varies

Nella tabella 2 vengono elencati i valori numerici delle metriche calcolate.

Matrix Name	**Time (ms)**	**Flops (GF)**	**SpeedUp (%)**
cage4	0.0115	0.545391	52.1739
olm1000	0.0575	8.89544	1657.39
west2021	0.1201	7.83667	1740.22
mhda416	0.0974	11.2519	1420.94
adder_dcop_32	14.3032	0.100641	1337.97
mcfe	0.432	7.2243	1450.93
rdist2	1.2197	5.97487	1119.95
cavity10	1.1559	8.45659	966.606
mhd4800a	1.029	12.7194	1023.13
bcsstk17	10.3934	5.27904	1261.73
raefsky2	2.5396	14.832	1096.79
thermal1	7.7252	9.51828	935.665
af23560	3.3422	18.5461	1058.28
thermomech_TK	15.8613	5.74224	891.257
olafu	9.824	13.2268	1119.76
FEM_3D_thermal1	3.199	17.235	954.204
lung2	7.1786	8.7828	977.489
amazon0302	20.9529	7.54379	991.982
roadNet-PA	77.8654	5.06934	1022.45
cop20k_A	59.3408	5.66077	1296.85
mac_econ_fwd500	55.1838	2.95365	1129.28
cant	33.4522	15.3337	1117.52
thermal2	123.977	8.85871	950.193
nlpkkt80	173.569	21.1685	1217.82
PR02R	84.6522	12.3765	1307.19
af_1_k101	86.8284	25.8727	1401.97
ML_Laplace	185.723	19.0839	1190.56
Cube_Coup_dt0	657.681	24.7573	1674.01

Table 2: OpenMP ELLPACK metrics with 20 threads and k = 64

5.2 Analisi CUDA

5.2.1 Formato CSR

I grafici 9 e 10 mostrano l'andamento dei *GigaFlops* e lo speedup al variare del numero di colonne del multivettore. Possiamo trarre le stesse conclusioni ottenute nel caso *OpenMP*: la differenza principale è data dall'elevata potenza di calcolo della GPU che consente di ottenere valori di *GFLOPS* molto più elevati rispetto al caso CPU e, di conseguenza, anche lo speedup subisce un miglioramento notevole.

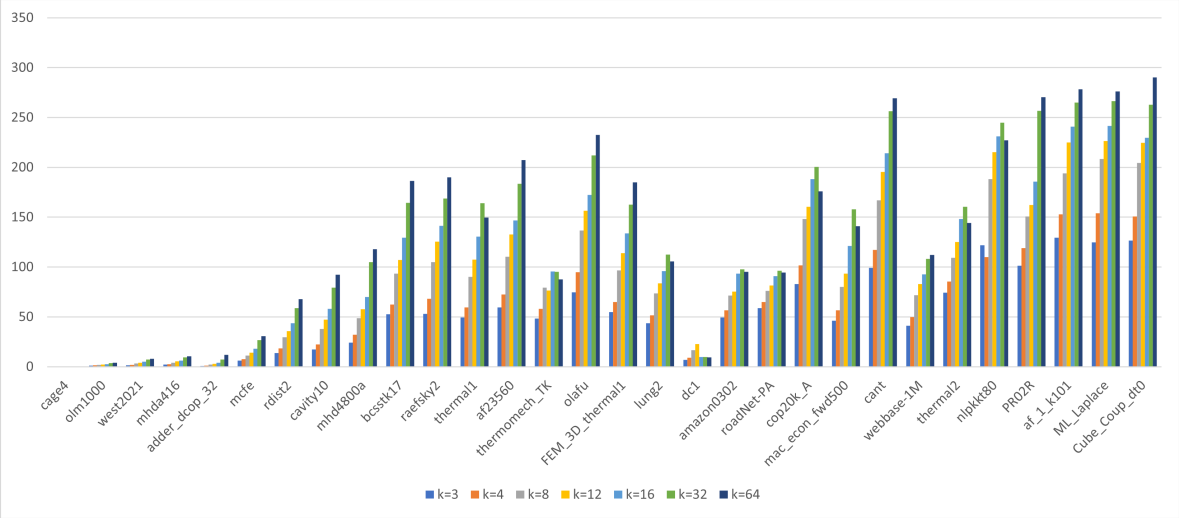

Chart 9: CUDA CSR GFLOPS as k varies

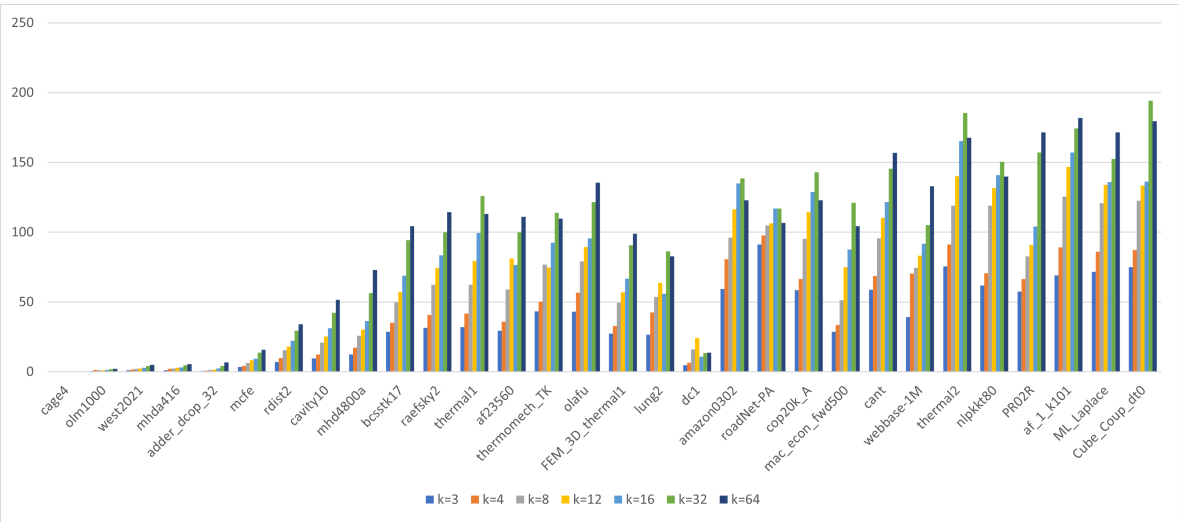

Chart 10: CUDA CSR speedup as k varies

La tabella 3 contiene i valori numerici delle metriche calcolate nel caso *CUDA* utilizzando il formato *CSR*.

Matrix Name	**Time (ms)**	**Flops (GF)**	**SpeedUp (%)**
cage4	0.0166144	0.377504	24.0755
olm1000	0.133888	3.82027	211.371
west2021	0.118333	7.9537	493.523
mhda416	0.104253	10.5123	535.237
adder_dcop_32	0.119414	12.0546	685.01
mcfe	0.102346	30.4937	1572.12
rdist2	0.10727	67.9363	3405.41
cavity10	0.105926	92.2808	5164.91
mhd4800a	0.11121	117.69	7298.83
bcsstk17	0.294538	186.282	10444.5
raefsky2	0.19831	189.941	11440.1
thermal1	0.491069	149.736	11319.4
af23560	0.298938	207.35	11101
thermomech_TK	1.04117	87.4776	10977.3
olafu	0.559126	232.398	13545.8
FEM_3D_thermal1	0.298298	184.831	9889.79
lung2	0.59767	105.49	8280.65
dc1	10.3166	9.50879	1376.06
amazon0302	1.65826	95.3196	12291.8
roadNet-PA	4.17287	94.5934	10671.8
cop20k_A	1.90972	175.897	12285.1
mac_econ_fwd500	1.15593	141.006	10441.4
cant	1.90526	269.226	15675.5
webbase-1M	3.54168	112.237	13303.7
thermal2	7.61803	144.169	16757
nlpkkt80	16.1886	226.962	13980.6
PR02R	3.87731	270.212	17164
af_1_k101	8.07742	278.119	18175.9
ML_Laplace	12.8281	276.293	17164.4
Cube_Coup_dt0	56.1441	290.01	17956.3

Table 3: CUDA CSR metrics with k = 64

5.2.2 Formato ELLPACK

Anche in questo caso i grafici che rappresentano l'andamento dei *GFLOPS* 11 e dello speedup 12 confermano i risultati ottenuti nel caso *OpenMP*. La differenza principale la vediamo nei valori di `k` più grandi, con i quali si ottengono le prestazioni migliori possibili per tutte le matrici. Ovviamente, le prestazioni ottenute per il formato *ELLPACK* nel caso *CUDA* sono notevolmente maggiori rispetto al caso *OpenMP* sia in termini di *GFLOPS* che in termini di speedup. Possiamo confermare, come nel caso *OpenMP*, il fatto che con il formato *CSR* si ottengono in generale prestazioni maggiori rispetto ad *ELLPACK*.

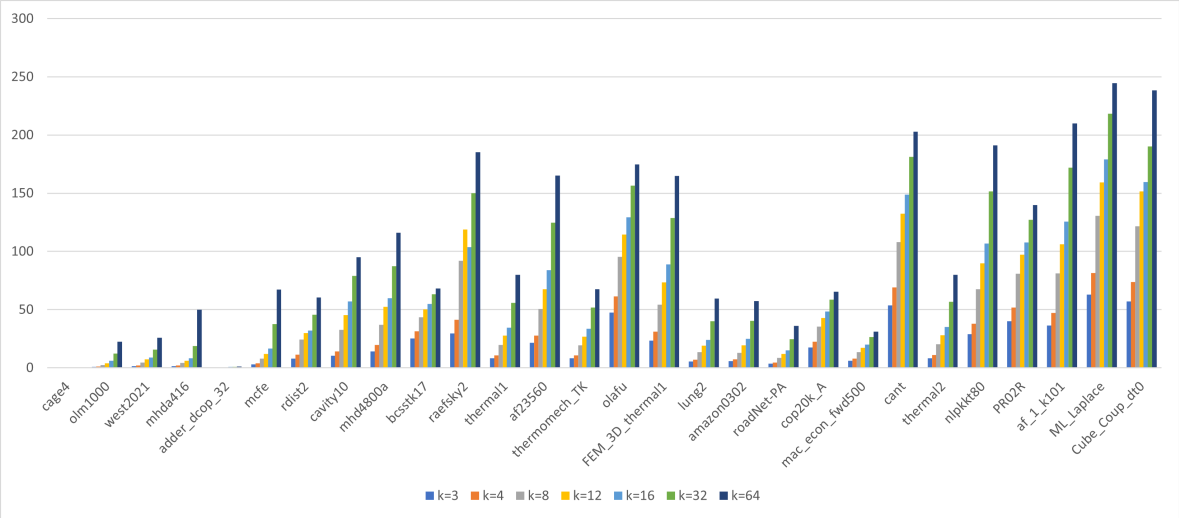

Chart 11: CUDA ELLPACK GFLOPS as k varies

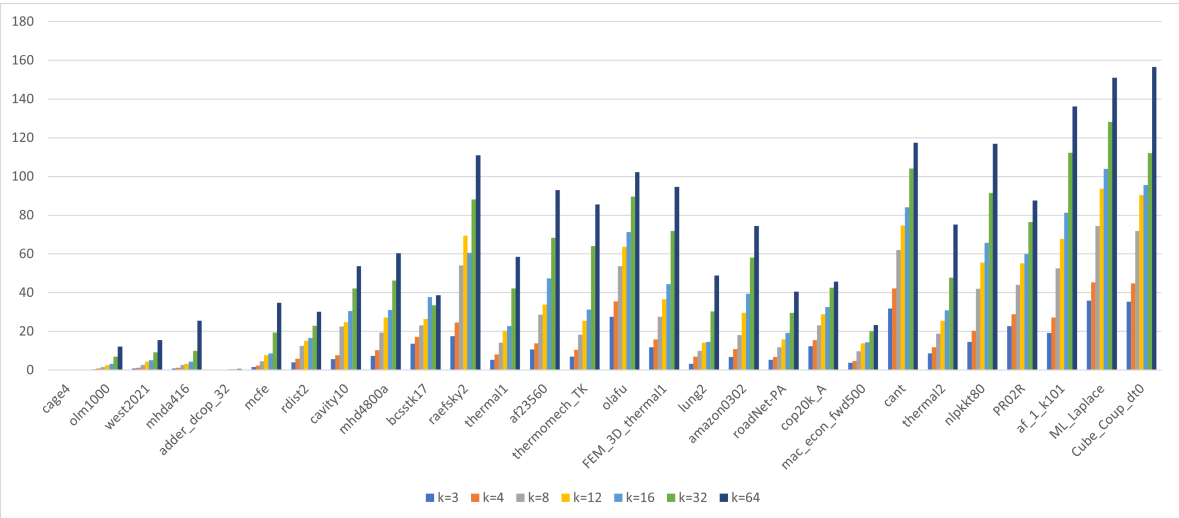

Chart 12: CUDA ELLPACK speedup as k varies

Nella tabella 4 sono elencati i valori numerici delle metriche calcolate nel caso *CUDA* utilizzando il formato *ELLPACK*.

Matrix Name	**Time (ms)**	**Flops (GF)**	**SpeedUp (%)**
cage4	0.023008	0.272601	13.0389
olm1000	0.022848	22.3866	1221.11
west2021	0.0366464	25.6829	1552.68
mhda416	0.0219712	49.8806	2557.89
adder_dcop_32	1.35238	1.06441	60.4861
mcfe	0.046512	67.0987	3482.97
rdist2	0.120688	60.3834	3017.7
cavity10	0.10295	94.9484	5360.83
mhd4800a	0.112778	116.054	6030.45
bcsstk17	0.805558	68.1108	3862.29
raefsky2	0.203363	185.222	11100.3
thermal1	0.920397	79.8901	5841.07
af23560	0.375331	165.147	9295.79
thermomech_TK	1.34593	67.6703	8554.98
olafu	0.743136	174.854	10232.7
FEM_3D_thermal1	0.334243	164.954	9459.28
lung2	1.05725	59.6341	4888.72
amazon0302	2.74958	57.4867	7441.9
roadNet-PA	10.9897	35.9177	4048.77
cop20k_A	5.14605	65.2761	4576.63
mac_econ_fwd500	5.21845	31.2341	2323.45
cant	2.52738	202.956	11747.2
thermal2	13.744	79.9095	7519.94
nlpkkt80	19.2175	191.19	11689.6
PR02R	7.48718	139.932	8769.99
af_1_k101	10.7049	209.855	13613.2
ML_Laplace	14.4839	244.707	15109.9
Cube_Coup_dt0	68.2634	238.523	15654.2

Table 4: CUDA ELLPACK metrics k = 64

5.2.3 Analisi prestazioni Nvidia Quadro RTX 5000

Siccome il prodotto *SpMM* è un problema *memory bound* è necessario conoscere la larghezza di banda massima della GPU utilizzata che è pari a **448 GB/s**. L'Intensità aritmetica massima del problema è circa **0.75** per cui ci possiamo aspettare che la velocità massima raggiungibile non sia maggiore di **336 GF**. Le prestazioni ottenute per le matrici di dimensioni maggiori oscillano tra i **100 GF** e i **300 GF** per il formato *CSR*, mentre si raggiungono valori compresi tra **20 GF** e **250 GF** per il formato *ELLPACK*. Questo risultato ci permette di concludere che è possibile migliorare il codice prodotto, nonostante le prestazioni ottenute abbastanza buone.

6 Conclusioni

Dall'analisi dei risultati ottenuti è possibile concludere che:

- Il formato *CSR* consente di ottenere prestazioni in generale maggiori rispetto a quello *ELLPACK*, anche se per alcune matrici questo non vale.
- Il prodotto *SpMM* eseguito su GPU consente di ottenere prestazioni di molto superiori rispetto ad eseguirlo su CPU, sopratutto con multivettori che hanno un numero di colonne più elevato.
- I fattori principali che impattano sulle prestazioni del programma sono le dimensioni della matrice e del multivettore e, soprattutto, avere una o più righe con troppi valori non-zero rispetto alle altre in una matrice.

Riferimenti bibliografici

[1] H. Zhang, R. T. Mills, K. Rupp, B. F. Smith. Vectorized parallel sparse matrix-vector multiplication in petsc using avx-512, 2018. URL: https://drive.google.com/file/d/1LCwkYecIpwd-rSVo1wp6CBac6ZphujDi/view.

[2] J. L. Greathouse, M. Daga. Efficient sparse matrix-vector multiplication on gpus using the csr storage format, 2014. URL: http://www.computermachines.org/joe/pdfs/sc2014_csr-adaptive.pdf.

[3] NVIDIA. Using cuda warp-level primitives, 2018. URL: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/.